Anchor: A Versatile and Efficient Framework for Resource Management in the Cloud

Hong Xu, *Student Member, IEEE,* Baochun Li, *Senior Member, IEEE*

Abstract—We present *Anchor*, a general resource management architecture that uses the stable matching framework to decouple policies from mechanisms when mapping virtual machines to physical servers. In *Anchor*, clients and operators are able to express a variety of distinct resource management policies as they deem fit, and these policies are captured as preferences in the stable matching framework. The highlight of *Anchor* is a new many-to-one stable matching theory that efficiently matches VMs with heterogeneous resource needs to servers, using both offline and online algorithms. Our theoretical analyses show the convergence and optimality of the algorithm. Our experiments with a prototype implementation on a 20-node server cluster, as well as large-scale simulations based on real-world workload traces, demonstrate that the architecture is able to realize a diverse set of policy objectives with good performance and practicality.

Index Terms—Cloud computing, resource management, stable matching, VM placement.

1 Introduction

Due to the multi-tenant nature, resource management becomes a major challenge for the cloud. According to a 2010 survey [1], it is the second most concerned problem that CTOs express after security. Cloud operators have a variety of distinct resource management objectives to achieve. For example, a public cloud such as Amazon may wish to use a workload consolidation policy to minimize its operating costs, while a private enterprise cloud may wish to adopt a load balancing policy to ensure quality of service. Further, VMs of a cloud also impose diverse resource requirements that need to be accommodated, as they run completely different applications owned by individual clients.

On the other hand, the infrastructure is usually managed as a whole by the operator, who relies on a single resource management substrate. Thus, the substrate must be *general* and *expressive* to accommodate a wide range of possible policies for different use cases, and be easily customizable and extensible. It also needs to be *fair* to orchestrate the needs and interests of both the operator and clients. This is especially important for private and federated clouds [2] where the use of money may not be appropriate to share resources fairly. Last but not the least, the resource management algorithm needs to be *efficient* so that large-scale problems can be handled.

Existing solutions fall short of the requirements we outlined. First, they tightly couple policies with mechanisms. Resource management tools developed by the industry such as VMware vSphere [3] and Eucalyptus [4], and by the open source community such as Nimbus [5] and CloudStack [6], do not provide support for configurable policies for VM placement. Existing papers on cloud resource management develop solutions for specific scenarios and purposes, such as consolidation based on CPU usage [7]–[9], energy consumption [10]–[12], bandwidth multiplexing [13]–[15], and storage dependence [16]. Moreover, these solutions are developed for the operator without considering the interest of clients.

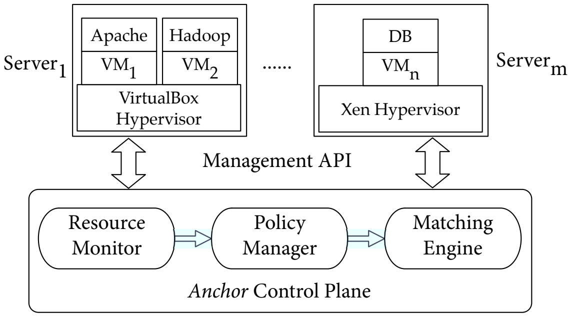

Fig. 1. The *Anchor* architecture.

We make three contributions in developing a versatile and efficient resource management substrate in the cloud. First, we present *Anchor*, a new architecture that decouples *policies* from *mechanisms* for cloud resource management. This is analogous to the design of BGP [17], where ISPs are given the freedom to express their policies, and the routing mechanism is able to efficiently accommodate them. *Anchor* consists of three components: a resource monitor, a policy manager, and a matching engine, as shown in Fig. 1. Both the operator and its clients are able to configure their resource management policies, based on performance, cost, etc., as they deem fit via the policy manager. When VM placement requests arrive, the policy manager polls information from the resource monitor, and feeds it with the policies to the matching engine. The matching mechanism resolves conflicts of interest among stakeholders, and outputs a matching between VMs and servers.

The challenge of *Anchor* is then to design an expressive,

- *The authors are with The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario M5S 3G4, Canada. Email: {henryxu, bli}@eecg.toronto.edu.*

fair, and efficient matching mechanism as we discussed. Our second major contribution is a novel matching mechanism based on the *stable matching* framework [18] from economics, which elegantly achieves all the design objectives. Specifically, the concept of *preferences* is used to enable stakeholders to express various policies with simple rank-ordered lists, fulfilling the requirement of generality and expressiveness. Rather than optimality, *stability* is used as the central solution concept to address the conflicts of interest among stakeholders, fulfilling the fairness requirement. Finally, its algorithmic implementations based on the classical *deferred acceptance* algorithm have been demonstrated to be practical in many real-world applications [18], fulfilling the efficiency requirement.

It may be tempting to formulate the matching problem as an optimization over certain utility functions, each reflecting a policy goal. However, optimization suffers from two important deficiencies in this case. First, as system-wide objectives are optimized, the solutions may not be appealing to clients, whose interest do not necessarily align well with the operator's. In this regard, a cloud resembles a resource market in which clients and the operator are autonomous selfish agents. *Individual rationality* needs to be respected for the matching to be acceptable to all participants. Second, optimization solvers are computationally expensive due to their combinatorial nature, and do not scale well.

The novelty of our stable matching mechanism lies in a rigorous treatment of size heterogeneity in Sec. 4. Specifically, classical stable matching theory cannot be directly applied here. Each VM has a different "size," corresponding to its demand for CPU, memory, and storage resources. Yet the economics literature assumes that each agent is uniform in size. Size heterogeneity makes the problem much more difficult, because even the very definition of stability becomes unclear in this case. We formulate a general *job-machine stable matching* problem with size heterogeneous jobs. We clarify the ambiguity of the conventional stability definition in our model, propose a new stability concept, develop algorithms to efficiently find stable matchings with respect to the new definition, and prove convergence and optimality results.

Our third contribution is a realistic performance evaluation of *Anchor*. We design a simple policy interface, and showcase several common policy examples in Sec. 5. We present a prototype implementation of *Anchor* on a 20-node server cluster, and conduct detailed performance evaluation using both experiments and large-scale simulations based on real-world workload traces in Sec. 6.

2 Background and Model

2.1 A Primer on Stable Matching

We start by introducing the classical theory of stable matching in the basic one-to-one marriage model [19]. There are two disjoint sets of agents, $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ and $\mathcal{W} = \{w_1, w_2, \ldots, w_p\}$, men and women. Each agent has a transitive preference over individuals on the other side, and the possibility of being unmatched [18]. Preferences can be represented as rank order lists of the form $p(m_1) = w_4, w_2, \ldots, w_i$, meaning that man m_1's first choice of partner is w_4, second choice is w_2 and so on, until at some point he prefers to be unmatched (i.e. matched to the empty set). We use $\succ_i$ to denote the ordering relationship of agent i (on either side of the market). If i prefers to remain unmatched instead of being matched to agent j, i.e. $\emptyset \succ_i j$, j is said to be *unacceptable* to i, and preferences can be represented just by the list of acceptable partners.

***Definition* 1:** An outcome is a *matching* $\mu : \mathcal{M} \times \mathcal{W} \times \emptyset \rightarrow \mathcal{M} \times \mathcal{W} \times \emptyset$ such that $w = \mu(m)$ if and only if $\mu(w) = m$, and $\mu(m) \in \mathcal{W} \cup \emptyset$, $\mu(w) \in \mathcal{M} \cup \emptyset$, $\forall m, w$.

It is clear that we need further criteria to distill a "good" set of matchings from all the possible outcomes. The first obvious criterion is *individual rationality*.

***Definition* 2:** A matching is *individual rational* to all agents, if and only if there does not exist an agent i who prefers being unmatched to being matched with $\mu(i)$, i.e., $\emptyset \succ_i \mu(i)$.

This implies that for a matched agent, its assigned partner should rank higher than the empty set in its preference. Between a pair of matched agents, they are not unacceptable to each other.

The second natural criterion is that a *blocking set* should not occur in a good matching:

***Definition* 3:** A matching μ is *blocked* by a pair of agents (m, w) if they each prefer each other to the partner they receive at μ. That is, $w \succ_m \mu(m)$ and $m \succ_w \mu(w)$. Such a pair is called a *blocking pair* in general.

When a blocking pair exists, the agents involved have a natural incentive to break up and form a new marriage. Therefore such an "unstable" matching is undesirable.

***Definition* 4:** A matching μ is *stable* if and only if it is individual rational, and not blocked by any pair of agents.

***Theorem* 1:** A stable matching exists for every marriage market.

This can be readily proved by the classic *deferred acceptance algorithm (DA)*, or the *Gale-Shapley algorithm* [19]. It works by having agents on one side of the market, say men, propose to the other side, in order of their preferences. As long as there exists a man who is free and has not yet proposed to every woman in his preference, he proposes to the most preferred woman who has not yet rejected him. The woman, if free, "holds" the proposal instead of directly accepting it. In case she already has a proposal at hand, she rejects the less preferred. This continues until no proposal can be made, at which point the algorithm stops and matches each woman to the man (if any) whose proposal she is holding. The woman-proposing version works in the same way by swapping the roles of man and woman. It can be readily seen that the order in which men propose is immaterial to the outcome.

2.2 Models and Assumptions

In a cloud, each VM is allocated a slice of resources from its hosting server. In this paper, we assume that the size of a slice is a multiple of an *atomic* VM. For instance, if the atomic VM has one CPU core equivalent to a 2007 Intel Xeon 1 GHz core, one memory unit equivalent to 512 MB PC-10600 DDR3 memory, and one storage unit equivalent to 10 GB 5400 RPM HDD, a VM of size 2 means it effectively has a 2 GHz 2007 Xeon CPU core, 1 GB PC-10600 DDR3 memory and 20 GB 5400 RPM hard disk. Note that the actual amount of resources is relative to the heterogeneous server hardware. Two VMs have the same size as long as performance is equivalent for all resources.

This may seem an oversimplification and raise concerns about its validity in reality. We comment that, in practice, such atomic sizing is common among large-scale public clouds to reduce the overhead of managing hundreds of thousands of VMs. It is also valid in production computer clusters [20], and widely adopted in related work [16], [21] to reduce the dimensionality of the problem. Appendix A provides more discussion on the validity of this assumption, especially with different job requirements.

We design *Anchor* for a setting where the workloads and resources demands of VMs are relatively stable. Resource management in the cloud can be naturally cast as a *stable matching* problem, where the overall pattern of common and conflicting interests between stakeholders can be resolved by confining our attention to outcomes that are stable. Broadly, it can be modelled as a *many-to-one* problem [19] where one server can enroll multiple VMs but one VM can only be assigned to one server. Preferences are used as an abstraction of policies no matter how they are defined.

In traditional many-to-one problems such as college admissions [19], each college has a quota of the number of students it can take. This cannot be directly applied to our scenario, as each VM has a different "size" corresponding to its demand for resources. We cannot simply define the quota of a server as the number of VMs it can take.

We formulate VM placement as a *job-machine stable matching* problem with size heterogeneous jobs. Each job has a size, and each machine has a *capacity*. A machine can host multiple jobs as long as the total job size does not exceed its capacity. Each job has a preference over all the acceptable machines that have sufficient capacities. Similarly, each machine has a preference over all the acceptable jobs whose size is smaller than its capacity. This is a more general many-to-one matching model in that the college admissions problem is a special case with uni-size jobs (students).

3 Theoretical Challenges of Job-Machine Stable Matching

We present theoretical challenges introduced by size heterogeneous jobs in this section.

Following convention, we can naturally define a *blocking pair* in job-machine stable matching based on the following intuition. In a matching μ, whenever a job j prefers a machine m to its assigned machine $\mu(j)$ (can be $\emptyset$ meaning it is unassigned), and m has vacant capacity to admit j, or when m does not have enough capacity, but by rejecting some or all of the accepted jobs that rank lower than j it is able to admit j, then j and m have a strong incentive to deviate from μ and form a new matching. Therefore,

***Definition* 5:** A job-machine pair (j, m) is a *blocking pair* if any of the two conditions holds:

$$\text{(a):}\qquad c(m) \geq s(j), j \succ_m \emptyset, \text{ and } m \succ_j \mu(j), \qquad (1)$$

$$\text{(b):}\qquad c(m) < s(j), c(m) + \sum_{j'} s(j') \geq s(j),$$

$$\text{where } j' \prec_m j, j' \in \mu(m), \text{ and } m \succ_j \mu(j). \qquad (2)$$

$c(m)$ denotes the capacity of machine m, and $s(j)$ denotes the size of job j.

Depending on whether a blocking pair satisfies condition (1) or (2), we say it is a *type-1* or *type-2* blocking pair. For example, in a setting shown in Fig. 2, the matching $A - (a), B - \emptyset$ contains two type-1 blocking pairs (b, B) and (c, B), and one type-2 blocking pair (c, A).

***Definition* 6:** A job-machine matching is *strongly stable* if it does not contain any blocking pair.

3.1 Non-existence of Strongly Stable Matchings

It is clear that both types of blocking pairs are undesirable, and we ought to find a strongly stable matching. However, such a matching may not exist in some cases. Fig. 2 shows one such example with three jobs and two machines. It can be verified that every possible matching contains either type-1 or type-2 blocking pairs.

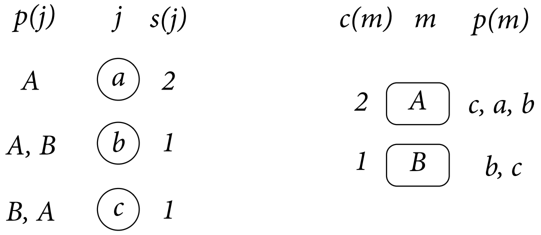

Fig. 2. A simple example where there is no strongly stable matching. Recall that $p()$ denotes the preference of an agent.

***Proposition* 1:** Strongly stable matching does not always exist.

Note that the definitions of type-1 and type-2 blocking pair coincide in classical problems with uni-size jobs. The reason why they do not remain so in our model is the size complementariness among jobs. In our problem, the concept of capacity denotes the amount of resources a machine can provide, which may be used to support different numbers of jobs. A machine's preferable job, which is more likely to be admitted in order to avoid type-2 blocking pairs, may consume less resources, and creates a higher likelihood for type-1 blocking pairs to happen on the same machine.

The non-existence result demonstrates the theoretical difficulty of the problem. We find that it is hard to even determine the necessary or sufficient conditions for the existence of strongly stable matchings in a given problem instance, albeit its definition seems natural. Therefore, for mathematical tractability, in the subsequent parts of the paper, we work with the following relaxed definition:

***Definition* 7:** A matching is *weakly stable* if it does not contain any type-2 blocking pair.

For example in Fig. 2, $A-(c), B-(b)$ is a weakly but not strongly stable matching, because it has a type-1 blocking pair (b, A). Thus, weakly stable matchings are a superset that subsumes strongly stable matchings. A matching is thus called *unstable* if it is not weakly stable.

3.2 Failure of the DA Algorithm

With the new stability concept, the first theoretical challenge is how to find a weakly stable matching, and does it always exist? If we can devise an algorithm that produces a weakly stable solution for any given instance, then its existence is clear. One may think that the deferred acceptance (DA) algorithm can be applied for this purpose. Jobs propose to machines following the order in their preferences. We randomly pick any free job that has not proposed to every machine on its preference to propose to its favorite machine that has not yet rejected it. That machine accepts the most favorable offers made so far up to the capacity, and rejects the rest. Unfortunately, we show that this may fail to be effective. Appendix B.1 shows such an example.

Two problems arise when applying the classical DA algorithm here. First, the execution sequence is no longer immaterial to the outcome. Second, it may even produce an unstable matching. This creates considerable difficulties since we cannot determine which proposing sequence yields a weakly stable matching for an arbitrary problem.

Examined more closely, the DA algorithm fails precisely due to the size heterogeneity of jobs. Recall that a machine will reject offers only when its capacity is used up. In the traditional setting with jobs of the same size, this ensures that whenever an offer is rejected, it must be the case that the machine's capacity is used up, and thus any offer made from a less preferred job will never be accepted, i.e. the outcome is stable. However, rejection due to capacity is problematic in our case, since a machine's remaining capacity may be increased, and its previously rejected job may become favorable again.

3.3 Optimal Weakly Stable Matching

There may be many weakly stable matchings for a problem instance. The next natural question to ask is then, which one should we choose to operate the system with? Based on the philosophy that a cloud exists for companies to ease the pain of IT investment and management, rather than the other way around, it is desirable if we can find a *job-optimal* weakly stable matching, in the sense that every job is assigned its best machine possible in all stable matchings.

The original DA algorithm is again not applicable in this regard, because it may produce type-1 blocking pairs even when the problem admits strongly stable matchings. Thus, our second challenge is to devise an algorithm that yields the job-optimal weakly stable matching. This quest is also theoretically important in its own right.

However, as we will show in Sec. 4.2, the complexity of solving this challenge is high, which may prevent its use in large-scale problems. Thus in many cases, a weakly stable matching is suitable for practical purposes.

4 A New Theory of Job-Machine Stable Matching

In this section we present our new theory of job-machine stable matching that addresses the above challenges.

4.1 A Revised DA Algorithm

We first propose a revised DA algorithm, shown in Table 1, that is guaranteed to find a weakly stable matching for a given problem. The key idea is to ensure that, whenever a job is rejected, any less preferable jobs will not be accepted by a machine, even if it has enough capacity to do so.

TABLE 1
Revised DA

```
Input: $c(m), p(m), \forall m \in \mathcal{M}, s(j), p(j), \forall j \in \mathcal{J}$
Initialize all $j \in \mathcal{J}$ and $m \in \mathcal{M}$ to free
while $\exists j$ who is free, and $p(j) \neq \emptyset$ do
    $m$ = $j$'s highest ranked machine in $p(j)$
    if $c(m) \geq s(j)$ then
        $j$ and $m$ become matched, $c(m) = c(m) - s(j)$
    else
        Find all $j'$ matched to $m$ so far such that $j' \prec_m j$
        repeat
            $m$ sequentially rejects each $j'$ by setting it to free, in the order of $p(m)$
            $c(m) = c(m) + s(j')$, best_rejected $= j'$
        until $c(m) \geq s(j)$ or all $j'$ are rejected
        if $c(m) \geq s(j)$ then
            $j$ and $m$ become matched, $c(m) = c(m) - s(j)$
        else
            $j$ becomes free, best_rejected = $j$
        for $j'' \in p(m), j'' \prec_m$ best_rejected do
            Remove $m$ from $p(j'')$, $j''$ from $p(m)$
Return: the final matching, and remaining capacity $c(m), \forall m \in \mathcal{M}$
```

The algorithm starts with a set of jobs $\mathcal{J}$ and a set of machines $\mathcal{M}$. Each job and machine are initialized to be free. Then the algorithm enters a propose-reject procedure. Whenever there are free jobs that have machines to propose to, we randomly pick one, say j, to propose to its current favorite machine m in $p(j)$, which contains all the machines that have not yet rejected it. If m has sufficient capacity, it holds the offer. Otherwise, it sequentially rejects offers from less preferable jobs j' until it can take the offer, in the order of its preference. If it still cannot do so even after rejecting all the j's, j is then rejected. Whenever a machine rejects a job, it updates the best_rejected variable, and at the end *all* jobs ranked lower than best_rejected are removed from its preference. The machine is also removed from preferences of all these jobs, as it will never accept their offers.

A pseudo-code implementation is shown in Table 1. We can see that the order in which jobs propose is immaterial, similar to the original DA algorithm. Moreover, we can prove that the algorithm guarantees that type-2 blocking pairs do not exist in the result.

***Theorem* 2:** The order in which jobs propose is of no consequence to the outcome in `Revised DA`.

***Theorem* 3:** `Revised DA`, in any execution order, produces a unique weakly stable matching.

Proof: The proof of uniqueness is essentially the same as that for the classical DA algorithm in the seminal paper [19]. We prove the weak stability of the outcome by contradiction. Suppose that `Revised DA` produces a matching μ with a type-2 blocking pair (j, m), i.e. there is at least one job j' worse than j to m in $\mu(m)$. Since $m \succ_j \mu(j)$, j must have proposed to m and been rejected. When j was rejected, j' was either rejected before j, or was made unable to propose to m because m is removed from the preferences of all the jobs ranked lower than j. Thus $j' = \emptyset$, which contradicts with the assumption. □

Theorem 3 also proves the existence of weakly stable matchings, as `Revised DA` terminates within $O(|\mathcal{J}|^2)$ in the worst case.

***Theorem* 4:** A weakly stable matchings exists for every job-machine matching problem.

The significance of `Revised DA` is multi-fold. It solves our first technical challenge in Sec. 3.2, and is appealing for practical use. The complexity is low compared to optimization algorithms. Further, it serves as a basic building block, upon which we develop an iterative algorithm to find the job-optimal weakly stable matching as we shall demonstrate soon. Lastly, it bears the desirable property of being insensitive to the order of proposing, which largely reduces the complexity of algorithm design.

`Revised DA` may still produce type-1 blocking pairs, and the result may not be job-optimal as defined in Sec. 3.3. In order to find the job-optimal matching, an intuitive idea is to run `Revised DA` multiple times, each time with type-1 blocking jobs proposing to machines that form blocking pairs with them. The intuition is that, type-1 blocking jobs can be possibly improved at no expense of others. However, simply doing so may make the matching unstable, because when a machine has both type-1 blocking jobs leaving from and proposing to it, it may have more capacity available to take jobs better than those it accepts according to its capacity before the jobs leaving. Readers may refer to Appendix B.2 for an example.

4.2 A Multi-stage DA Algorithm

We now design a *multi-stage* DA algorithm to iteratively find a better weakly stable matching with respect to jobs. The algorithm proceeds in stages. Whenever there is a type-1 blocking pair (j, m) in the result of previous stage μ_{t-1}, the algorithm enters the next stage where the blocking machine m will accept new offers. The blocking job j is removed from its previous machine $\mu_{t-1}(j)$, so that it can make new offers to machines that have rejected it before. $\mu_{t-1}(j)$'s capacity is also updated accordingly. Moreover, to account for the effect of job removal, all jobs that can potentially form type-1 blocking pairs with $\mu_{t-1}(j)$ if j leaves (there may be other machines that j form type-1 blocking pairs with) are also removed from their machines and allowed to propose in the next stage (corresponding to the while loop in step 7). This ensures that the algorithm does not produce new type-2 blocking pairs during the course, as we shall prove soon. At each stage, we run `Revised DA` with the selected set of proposing jobs $\mathcal{J}'$, and the entire set of machines with updated capacity $c^{pre}_t(m)$. The entire procedure is shown in Table 2.

TABLE 2
`Multi-stage DA`

```
1: Input: $c(m), p(m), \forall m \in \mathcal{M}$, $s(j), p(j), \forall j \in \mathcal{J}$.
2: $\mu_0 = \emptyset$, $t = 0$, stop = false, $\mathcal{J}' = \emptyset$
3: while stop == false do
4:     $t = t + 1$, $\mu' = \mu_{t-1}$
5:     for $m \in \mathcal{M}$ do
6:         $c^{pre}_t(m) = c_{t-1}(m)$
7:     while $\Omega \neq \emptyset$, where $\Omega$ is the set of jobs that form type-1
       blocking pairs from $\mu'$ with $c^{pre}_t(m)$ do
8:         for $j \in \Omega$ do
9:             Add $j'$ to $\mathcal{J}'$.
10:            if $\mu'(j)$ != $\emptyset$ then
11:                $c^{pre}_t(\mu'(j)) = c^{pre}_t(\mu'(j)) + s(j)$.
12:                $j'$ is free and removed from the matching $\mu'$.
13:    if $\mathcal{J}'$ == $\emptyset$ then
14:        break
15:    ($\mu_t$, $c_t(m)$) = Revised DA($c^{pre}_t(m), p(m)$,
16:                            $s(j), p(j), \mu', \mathcal{J}'$)
17:    if $\mu_t$ == $\mu_{t-1}$ then
18:        stop = true
19: Return $\mu_t$
```

We now prove important properties of `Multi-stage DA`, namely its correctness, convergence, and job-optimality.

4.2.1 Correctness

First we establish the correctness of `Multi-stage DA`.

***Theorem* 5:** There is no type-2 blocking pair in the matchings produced at any stage in `Multi-stage DA`.

Proof: This can be proved by induction. As the base case, we already proved that there is no type-2 blocking pair after the first stage in Theorem 3.

Given there is no type-2 blocking pair after stage t, we need to show that after stage $t + 1$, there is still no type-2 blocking pair. Suppose after $t+1$, there is a type-2 blocking pair (j, m), *i.e.*, $c_{t+1}(m) < s(j)$, $c_{t+1}(m) + \sum_{j'} s(j') \geq s(j)$, where $j' \prec_m j, j' \in \mu_t(m), m \succ_j \mu_{t+1}(j)$. If $c^{pre}_{t+1}(m) \geq s(j)$, then j must have proposed to m and been rejected according to the algorithm. Thus it is impossible for m to accept any job j' less preferable than j in $t + 1$.

If $c^{pre}_{t+1}(m) < s(j)$, then j did not propose to m in $t + 1$. Since there is no type-2 blocking pairs after t, j' must be accepted in $t + 1$. Now since $c^{pre}_{t+1}(m) < s(j)$, the sum of the remaining capacity and total size of newly accepted jobs after $t + 1$ must be less than $c^{pre}_{t+1}(m)$, *i.e.* $c_{t+1}(m) + \sum_{j''} s(j'') \leq c^{pre}_{t+1}(m) < s(j)$, where j'' denotes the newly accepted jobs in $t+1$. This contradicts with the assumption that $c_{t+1}(m) + \sum_{j'} s(j') \geq s(j)$ since $\{j'\} \subseteq \{j''\}$.

If $c^{pre}_{t+1}(m) = 0$, then m only has jobs leaving from it. Since there is no type-2 blocking pair after t, clearly there cannot be any type-2 blocking pair in $t+1$.

Therefore, type-2 blocking pairs do not exist at any stage of the algorithm. The uniqueness of the matching result at each stage is readily implied from Theorem 3. □

4.2.2 Convergence

Next we prove the convergence of `Multi-stage DA`. The key observation is that it produces a weakly stable matching at least as good as that in the previous stage from the job's perspective.

***Lemma* 1:** At any consecutive stages t and $t+1$ of `Multi-stage DA`, $\mu_{t+1}(j) \succeq_j \mu_t(j), \forall j \in \mathcal{J}$.

Proof: Refer to Appendix C.1. □

Therefore, the algorithm always tries to improve the weakly stable matching it found in the previous stage, whenever there is such a possibility suggested by the existence of type-1 blocking pairs. However, Lemma 1 also implies that a job's machine at $t+1$ may remain the same as in the previous stage. In fact, it is possible that the entire matching is the same as the one in previous stage, i.e. $\mu_{t+1} = \mu_t$. This can be easily verified using the example of Fig. 2. After the first stage, the weakly stable matching is $A-(c), B-(b)$. First b wishes to propose to A in the second stage. Then we assign b to $\emptyset$ and B has capacity of 1 again. c then wishes to propose to B too. After we remove c from A and update A's capacity, a now wishes to propose to A. Thus at the next stage, the same set of jobs a, b, c will propose to the same set of machines with same capacity, and the result will be the same matching as in the first stage. In this case, `Multi-stage DA` will terminate with the final matching that it cannot improve upon as its output (step 17-18 of Table 2). We thus have:

***Theorem* 6:** `Multi-stage DA` terminates in finite time.

Note that in each stage, `Multi-stage DA` may result in new type-1 blocking pairs, and the number of type-1 blocking pairs is not monotonically decreasing. Thus its worst case run time complexity is difficult to analytically derive. In Sec. 6.3 we evaluate its average case complexity through large-scale simulations.

4.2.3 Job-Optimality

We now prove the most important result regarding `Multi-stage DA`:

***Theorem* 7:** `Multi-stage DA` always produces the *job-optimal* weakly stable matching when it terminates, in the sense that every job is at least as good in the weakly stable matching produced by the algorithm as it would be in any other weakly stable matching.

Proof: We provide a proof sketch here. A detailed proof can be found in Appendix C.2. The algorithm terminates at stage t when either there is no type-1 blocking pair, or there is type-1 blocking pair(s) but $\mu_t = \mu_{t-1}$. For the former case, we show that our algorithm only permanently rejects jobs from machines that are impossible to accept them in all weakly stable matchings, when the jobs cannot participate any further. The outcome is therefore optimal. For the latter case, we can also show that it is impossible for jobs that participated in t to obtain a better machine. □

TABLE 3
Anchor's policy interface.

Functionality	*Anchor* API Call
create a policy group	`g = create()`
add/delete server	`add/delete(g_o,s)`
add/delete VMs	`add/delete(g_c,v)`
set ranking factors	`conf(g,factor1,...)`
set placement constraints	`limit(g_c,servers)`
colocation/anti-colocation	colocate(tenants,i,*g_c*)

Finally, we present another fact regarding the outcome of our algorithm.

***Theorem* 8:** `Multi-stage DA` produces a unique job-optimal *strongly* stable matching when it terminates with no job proposing.

The proof can be found in Appendix C.3.

4.3 An Online Algorithm

We have thus far assumed a static setting with a fixed set of jobs and machines. In practice, requests for job (VM) placement arrive dynamically, and we need to make decisions on the fly. It may not be feasible to re-run the matching algorithm from scratch every time when there is a new job. We further develop an online algorithm based on `Revised DA` that handles the dynamic case efficiently. Interested readers can find the detailed algorithm design and evaluation results in Appendix D and F.3, respectively.

5 Showcases of Resource Management Policies with the Policy Engine

We have presented the underlying mechanism of *Anchor* that produces a weakly stable matching between VMs of various sizes, as *jobs*, and physical servers, as *machines*. We now introduce *Anchor*'s policy engine which constructs preference lists according to various resource management policies. The cloud operator and clients interact with the policy engine through an API as shown in Table 3.

In order to reduce management overhead, we use *policy groups* that can be created with the `create()` call. Each policy group contains a set of servers or VMs that are entitled to a common policy. In fact some of the recent industry products have adopted similar ideas to help companies manage their servers in the cloud [22]. The policy is configured by the `conf()` call that informs the policy engine what factors to be considered for ranking the potential partners in a descending order of importance. The exact definition of ranking factors varies depending on the specific policy as we demonstrate in the following. With policy groups, only one common preference list is needed for all members of the group. Membership is maintained by `add()` and `delete()` calls. `colocate()` and `limit()` are used to set colocation/anti-colocation and placement constraints as we discuss in Appendix E.

It is also possible for the operator to configure policies on behalf of its clients if they do not explicitly specify any. This is done by enrolling them to the `default` policy group.

5.1 Cloud Operators

We begin our discussion from the operator's perspective.

Server consolidation/packing. Operators usually wish to consolidate the workload by packing VMs onto a small number of highly occupied servers, so that idle servers can be powered down to save operational costs. To realize this policy, servers can be configured to prefer a VM with a larger size. This can be done using `conf(g_o, 1/vm_size)`, where `g_o` is the operator's policy group. For VMs in the `default` policy group, their preference is ranked in the descending order of server load. One may use the total size of active VMs as the metric of load (`conf(g_c, 1/server_load)`), where `g_c` is the client's policy group. Alternatively, the number of active VMs can also serve as a heuristic metric (`conf(g_c, 1/num_vm)`).

Notice that consolidation is closely related to packing, and the above configuration resembles the *first fit decreasing* heuristic widely used to solve packing problems by iteratively assigning the largest item to the first bin that fits.

Load balancing. Another popular resource management policy is load balancing, which distributes VMs across all servers to mitigate performance degradation due to application dynamics over time. This can be seen as the opposite of consolidation. In this case, we can configure the server preference with `conf(g_o, vm_size)`, implying that servers prefer smaller VMs in size. VMs in the `default` policy group are configured with `conf(g_c, server_load)`, such that they prefer less utilized servers. This can be seen as a *worst fit increasing* heuristic.

5.2 Cloud Clients

From the perspective of cloud clients, other than choosing to join the `default` policy group and follow the operator's configuration, they can also express their unique policies.

Resource hunting. Depending on the resource demand of applications, VMs can be CPU, memory, or bandwidth-bound, or even resource-intensive in terms of multiple resources. Though resources are sliced into fixed slivers, most modern hypervisors support dynamic resizing of VMs. For example, the hypervisor may allow a temporarily burst of CPU usage for a VM provided that doing so does not affect colocated VMs. For memory, with a technique known as *memory ballooning*, the hypervisor is able to dynamically reduce the memory footprints of idle VMs, so that memory allocation of heavily loaded VMs can be increased.

Thus, clients may configure their policies according to the resource usage pattern of their VMs, which is unknown to the operator. CPU-bound VMs can be added to a *CPU-bound* policy group, which is configured with a call to `conf(g_c, 1/server_freecpu)`. Their preferences are then ranked in the descending order of server's time average available CPU cycles. Similarly, memory-bound and bandwidth-bound policy groups may be configured with the call `conf(g_c, 1/server_freemem)` and `conf(g_c, 1/server_freebw)`, respectively.

Anchor supports additional policies besides what we list here, including colocation/anti-colocation, tiered services, etc. Due to space limit, readers are directed to Appendix E for more details.

6 Implementation and Evaluation

We investigate the performance of *Anchor* with both testbed implementation and large-scale simulations based on real-world workload traces.

6.1 Setup

Prototype implementation. Our prototype consists of about 1500 LOC written in Python. It is based on Oracle VirtualBox 3.2.10 [23]. The VirtualBox management API is utilized to obtain resource usage statistics. More details can be found in Appendix F.1.

Our evaluation of *Anchor* is based on a prototype data center consisting of 20 Dual Dual-Core Intel Xeon 3.0 GHz machines connected over gigabit ethernet. Each machine has 2 GB memory. Thus, we define the atomic VM to have 1.5 GHz CPU and 256 MB memory. Each server has a capacity of 7 in terms of atomic VM (since the hypervisor also consumes server resources). All machines run Ubuntu 8.04.4 LTS with Linux 2.6.24-28 server. A cluster of Dual Intel Xeon 2.4 Ghz servers are used to generate workload for some of the experiments. One node in the cluster is designated to run the *Anchor* control plane, while others host VMs. Our VMs, if not otherwise noted, run Ubuntu 8.10 server with Apache 2.2.9, PHP 5.2.6, and MySQL 5.0.67.

Trace-driven simulation. To evaluate *Anchor* at scale, we conduct large-scale simulation based on real-world workload traces from RICC (RIKEN Integrated Cluster of Clusters) [20] in Japan. RICC is composed of 4 clusters, and was put into operation in August 2009. The data provided in the trace is from the "massively parallel cluster," which has 1024 Fujitsu RX200S5 Cluster nodes, each with 12 GB memory and two 4-core CPUs, for a total of 12 TB memory and 8192 cores. The trace file contains workload during the period of Sat May 01 00:04:55 JST 2010, to Thu Sep 30 23:58:08 JST 2010.

6.2 Efficiency of Resource Allocation

We evaluate the efficiency of *Anchor* resource allocation, by allowing clients to use the resource hunting policy in Sec. 5.2. We enable memory ballooning in VirtualBox to allow the temporary burst of memory use. CPU-bound VMs are configured to run a 20 newsgroups Bayesian classification job with 20,000 newsgroups documents, based on the *Apache Mahout* machine learning library [24]. Memory-bound VMs run a Web application called *Olio* that allows users to add and edit social events and share with others [25]. Its MySQL database is loaded with a large amount of data so that performance is memory critical. We use *Faban*, a benchmarking tool for tiered web applications, to inject workload and measure Olio's performance [26].

Our experiment comprises of 2 servers (S1, S2) and 2 VMs (VM1 and VM2). S1 runs a memory-bound VM of size 5, and S2 runs a CPU-bound VM of the same size before allocation. VM1 is CPU-bound with size 1 while VM2 is memory-bound with size 2. Assuming servers adopt a consolidation policy, we run *Anchor* first with the resource hunting policy, followed by another run with the default consolidation policy for the two VMs. In the first run, *Anchor* matches VM1 to S1 and VM2 to S2, since VM1 prefers S1 with more available CPU and VM2 prefers S2 with more memory. Other VM placement schemes that consider the resource usage pattern of VMs will yield the same matching. In the second run, *Anchor* matches VM2 to S1 and VM1 to S2 for consolidation.

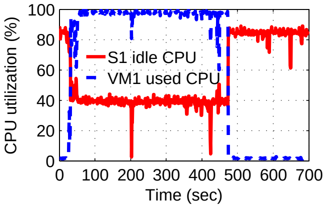

Fig. 3. VM1 CPU usage on S1 when using the resource hunting policy.

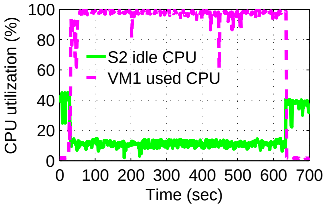

Fig. 4. VM1 CPU usage on S2 when using the consolidation policy.

We now compare CPU utilization of VM1 in these two matchings as shown in Fig. 3 and 4, respectively. From Fig. 3, we can see that as VM1 starts the learning task at around 20 seconds, it quickly hogs its allocated CPU share of 12.5%, and bursts to approximately 40% on S1 (80%-40%). Some may wonder why it does not saturate S1's CPU. We conjecture that the reason may be related to VirtualBox's implementation that limits the CPU allocated to a single VM. In the case it is matched to S2, it can only consume up to about 30% CPU, while the rest is taken by S2's pre-existing VM as seen in Fig. 4. We also observe that the learning task takes about 600 seconds to complete on S2, compared to only 460 seconds on S1, which implies a performance penalty of 30%.

We next look at the memory-bound VM2. Fig. 5 shows time series of memory allocation comparison between the two matchings. Recall that VM2 has size 2, and should be allocated 512 MB memory. By the resource hunting policy, it is matched to S2, and obtains its fair share as soon as it is started at around 10 seconds. When we start the Faban workload generator at 50 seconds, its memory allocation steadily increases as an effect of memory ballooning to cope with the increasing workload. At steady state it utilizes about 900 MB. On the other hand, when it is matched to S1 by the consolidation policy, it only has 400 MB memory after startup. The deficit of 112 MB is allocated to the other memory hungry VM that S1 is running. VM2 gradually reclaims its fair share as the workload of Olio database rises, but cannot get any extra resource beyond that point.

Client resource hunting policy also serves to the benefit of the operator and its servers. Fig. 6 shows S1's resource utilization. When resource hunting policy is used, *i.e.* when S1 is assigned VM1, its total CPU and memory utilization are aligned at around 60%, because VM1's CPU-bound nature is complementary to the memory-bound nature of S1's existing VM. However, when S1 is assigned the memory-bound VM2 by the consolidation policy, its memory utilization surges to nearly 100% while CPU utilization lags at only 50%. A similar observation can be made for S2.

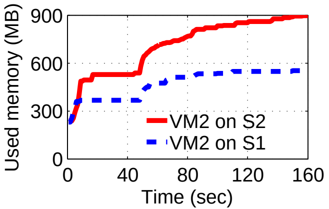

Fig. 5. VM2 memory usage on S1 and S2.

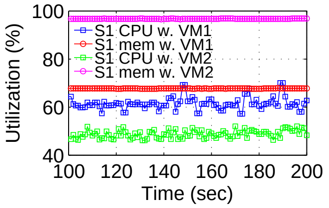

Fig. 6. S1 CPU and memory usage.

Result: Anchor *enables efficient resource utilization of the infrastructure and improves performance of its VMs, by allowing individual clients to express policies specific to its resource needs.*

6.3 Anchor's Performance at Scale

Now we evaluate the performance and scalability of *Anchor* using both experiments and trace-driven simulations. We first conduct a small-scale experiment involving placing 10 VMs to 10 servers using both the consolidation and load balancing policies. The results, as can be found in Appendix F.2, show that *Anchor* is effective in realizing specified policies in a small-scale setup.

We then conduct a medium-scale experiment involving all of our 20 machines. We orchestrate a complex scenario with 4 batches of VMs, each with 20 VMs whose sizes is drawn uniformly randomly in $[1, 4]$. Servers are initially empty with a capacity of 7. VMs are uniformly randomly chosen to use either consolidation, CPU-bound, or memory-bound resource hunting, and servers adopt a consolidation policy for placement.

Since the stakeholders have different objectives, we use the rank percentile of the assigned partner as the performance metric that reflects one's "happiness" about the matching. A 90% happiness then means that the partner ranks better than 90% of the total population. For servers, their happiness is the average of the matched VMs. From the experiment we find that VMs obtain their top 10% partner on average while servers only get their top 50% VMs. The reason is that the number of VMs is too small compared to servers' total capacity, and most of VMs' proposals can be directly accepted.

The scale of previous experiments is limited due to the hardware constraint of our testbed. To verify *Anchor*'s effectiveness in a practical cloud scenario with large numbers of VMs and servers, we perform large-scale trace-driven simulations using the RICC workload traces as the input to our `Revised DA` and `Multi-stage DA` algorithms. According to [20], the allocation of CPU and memory of this cluster is done with a fixed ratio of 1.2 GB per core, which coincides well with our atomic VM assumption. We thus define an atomic VM to be of 1 core with 1.2 GB memory. Each RICC server, with 8 cores and 12 GB memory as introduced in Sec. 6.1, has a capacity of 8. The number of servers is fixed at 1024.

We assume that tasks in the trace run in VMs, and they arrive offline before the algorithms run. We consider the dynamic scenario with our online algorithm in Appendix F.3. For large tasks that require more than one server, we break them down into multiple smaller tasks, each of size 8, that can run on a single server. We then process each task scheduling request in the trace as VM placement request(s) of various sizes. We use the first 200 tasks in the trace, which amounts to more than 1000 VM requests.

However, the trace does not have detailed information regarding the resource usage history of servers and tasks, making it difficult for us to generate various preferences needed for stable matching. To emulate a typical operational cloud with a few policy groups, we synthesize 8 policy groups for servers and 10 for VMs, the preference of each group being a random permutation of members of the other side. The results are averaged over 100 runs.

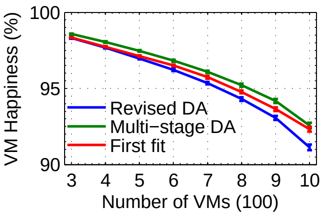

Fig. 7. VM happiness in a static setting.

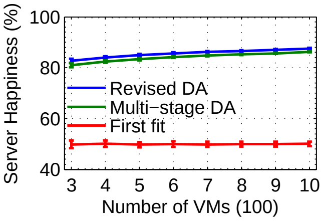

Fig. 8. Server happiness in a static setting.

As a benchmark, we implement a `First fit` algorithm widely used to solve large-scale VM placement problems in the literature [8], [9], [12]. Since the servers have different preferences but `First fit` algorithm assumes a uniform ranking of VMs, the algorithm sorts the VMs according to the preference of the most popular policy group first, and places a VM to the best server according to the VMs preference that has enough capacity.

Fig. 7 and 8 show the results with error bars for both `Revised DA` and `Multi-stage DA` with different scales. As expected, we observe that, as the problem scales up, VMs are allocated to lower ranked servers and their happiness decreases, and servers are allocated with higher ranked VMs, due to the increased competition amongst VMs. Also note that `Multi-stage DA` is only able to improve the matching from the VM perspective by 15% on average as shown in Fig. 7, at the cost of decreased server happiness as shown in Fig. 8. The performance difference between `Revised DA` and `Multi-stage DA` for VMs are thus small.

Compared to the benchmark `First fit`, our algorithms provide significant performance improvement for servers. Both `Revised DA` and `Multi-stage DA` consistently improve the server happiness by 60% for all problem sizes. This demonstrates the advantage of our algorithms in coordinating the conflicting interests between the operator and the clients using stable matching. Specifically, `First fit` only uses a single uniform ranking of VMs for all servers, while our stable matching algorithms allow servers to express their own preferences. Further, `First fit` will not match a VM to a server whose capacity is insufficient, i.e. there will be no rejection from servers, while `Online DA` allows rejections if a VM is preferable than some of the server's VMs during its execution. Clearly this improves the happiness of both VMs and servers.

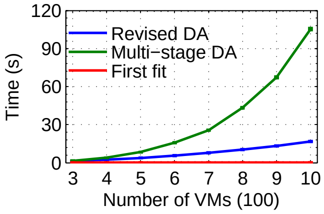

Fig. 9. Running time in the static setting.

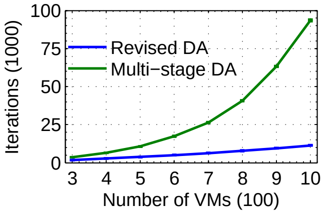

Fig. 10. Number of iterations in the static setting.

Fig. 9 and 10 show the time complexity of the algorithms. It is clear that the running time of `Multi-stage DA` is much worse than the simple `Revised DA`, and grows more rapidly. The same observation is made for the number of iterations, where `Multi-stage DA` takes more than 95000 iterations to finish while `Revised DA` takes only 11824 iterations with 1000 VMs. Another observation we emphasize here is that the average case complexity of `Revised DA` is much lower than its worst case complexity $O(|\mathcal{J}|^2)$ in Sec. 4.1, while `Multi-stage DA` exhibits $O(|\mathcal{J}|^2)$ complexity on average. Thus `Revised DA` scales well in practice, while `Multi-stage DA` may only be used for small or medium scale problems.

`Revised DA` takes 10 seconds to solve problems with 1000 VMs and 1024 servers, which is acceptable for practical use. As expected, both algorithms are slower than the simple `First fit` algorithm, whose running time is negligible (0.01s–0.06s). `First fit` is not iterative so we do not include it in Fig. 10 for the number of iterations comparison.

Result: `Revised DA` *is effective and practical for large-scale problems with thousands of VMs, and offers very close-to-optimal performance for VMs.*

7 Related Work

This work is related to research in the following fields.

Stable Matching. A large body of research in economics has examined variants of stable matching [19] (see [18], [27] and references therein). Algorithmic aspects of stable matching have also been studied in computer science [28], [29]. However, the use of stable matching in networking is fairly limited. [16] uses the DA algorithm to solve the coupled placement of VMs in datacenters. Our recent work [30], [31] advocate stable matching as a general framework to solve networking problems. To our knowledge, all these works assume a traditional uni-size job model, while we study a more general size-heterogeneous model.

VM Placement. VM placement on a shared infrastructure has been extensively studied. Current industry solutions from virtualization vendors such as VMware vSphere [3] and Eucalyptus [4], and open-source efforts such as Nimbus [5] and CloudStack [6], only provide a limited set of predefined placement policies. Existing papers develop specifically crafted algorithms and systems for specific scenarios, such as consolidation based on CPU usage [7]–[9], energy consumption [10]–[12], bandwidth multiplexing [13]–[15], and storage dependence [16]. They are thus complementary to *Anchor*, as the insights and strategies can be incorporated as policies to serve different purposes without the need to design new algorithms from the ground up.

OpenNebula [32], a resource management system for virtualized infrastructures, is the only related work to our knowledge that also decouples management policies with mechanisms. It uses a simple first fit algorithm based a configurable ranking scheme to place VMs, while we use the stable matching framework that addresses the conflict of interest between the operator and clients.

There is a small literature on online VM placement. [33]–[35] develop systems to predict the dynamic resource demand of VMs and guide the placement process. [15] considers minimizing the long-term routing cost between VMs. These works consider various aspects to refine the placement process and are orthogonal to our work that addresses the fundamental problem of VM size heterogeneity.

Our work is also related to the literature on job scheduling. More details can be found in Appendix G.

8 Concluding Remarks

We presented *Anchor* as a unifying fabric for resource management in the cloud, where *policies* are decoupled from the management *mechanisms* by the stable matching framework. We developed a new theory of job-machine stable matching with size heterogeneous jobs as the underlying mechanism to resolve conflict of interests between the operator and clients. We then showcased the versatility of the *preference* abstraction for a wide spectrum of resource management policies for VM placement with a simple API. Finally, the efficiency and scalability of *Anchor* are demonstrated using a prototype implementation and large-scale trace-driven simulations.

Many other problems can be cast into our model. For instance, job scheduling in distributed computing platforms such as MapReduce, where jobs have different sizes and share a common infrastructure. Our theoretical results are thus potentially applicable to scenarios beyond those described in this paper. As future work, we plan to extend *Anchor* for the case where resource demands vary, and VMs may require to be re-placed, where specific considerations for VM live migration [21] are needed.

References

[1] Zenoss Inc., "Virtualization and cloud computing survey," 2010.

[2] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojičić, D. O'Hallaron, and Y. C. Soh, "Open Cirrus™ cloud computing testbed: Federated data centers for open source systems and services research," in *Proc. USENIX HotCloud*, 2009.

[3] VMware vSphere™, http://www.vmware.com/products/drs/overview.html.

[4] Eucalyptus, http://www.eucalyptus.com.

[5] Nimbus, http://www.nimbusproject.org.

[6] CloudStack, http://www.cloudstack.org.

[7] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 239–254, December 2002.

[8] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *Proc. Intl. Conf. Autonomic Computing (ICAC)*, 2010.

[9] M. Chen, H. Zhang, Y.-Y. Su, X. Wang, G. Jiang, and K. Yoshihira, "Effective VM sizing in virtualized data centers," in *Proc. IFIP/IEEE Intl. Symp. Inte. Netw. Manag. (IM)*, 2011.

[10] R. Nathuji and K. Schwan, "Virtualpower: Coordinated power management in virtualized enterprise systems," in *Proc. SOSP*, 2007.

[11] A. Verma, P. Ahuja, and A. Neogi, "pMapper: Power and migration cost aware application placement in virtualized systems," in *Proc. ACM Middleware*, 2008.

[12] M. Cardosa, M. R. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server environments," in *Proc. IFIP/IEEE Intl. Symp. Inte. Netw. Manag. (IM)*, 2009.

[13] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, 2010.

[14] D. Breitgand and A. Epstein, "Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds," in *Proc. IEEE INFOCOM*, 2012.

[15] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proc. IEEE INFOCOM*, 2012.

[16] M. Korupolu, A. Singh, and B. Bamba, "Coupled placement in modern data centers," in *Proc. IPTPS*, 2009.

[17] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," RFC 4271, http://datatracker.ietf.org/doc/rfc4271/, January 2006.

[18] A. E. Roth, "Deferred acceptance algorithms: History, theory, practice, and open questions," *Int. J. Game Theory*, vol. 36, pp. 537–569, 2008.

[19] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *Amer. Math. Mon.*, vol. 69, no. 1, pp. 9–15, 1962.

[20] The RICC Log, http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html.

[21] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. NSDI*, 2007.

[22] http://gigaom.com/cloud/tier-3-spiffs-up-cloud-management/.

[23] Oracle VirtualBox, http://www.virtualbox.org.

[24] Apache Mahout, http://mahout.apache.org.

[25] Olio, http://incubator.apache.org/olio.

[26] Faban, http://www.opensparc.net/sunsource/faban/www.

[27] A. E. Roth and M. Sotomayor, *Two-sided Matching: A Study in Game Theoretic Modeling and Analysis*, ser. Econometric Society Monograph. Cambridge University Press, 1990, no. 18.

[28] R. Irving, P. Leather, and D. Gusfield, "An efficient algorithm for the "optimal" stable marriage," *J. ACM*, vol. 34, no. 3, pp. 532–543, 1987.

[29] D. F. Manlove, R. W. Irving, K. Iwama, S. Miyazaki, and Y. Morita, "Hard variants of stable marriage," *Elsevier Theoretical Computer Science*, vol. 276, pp. 261–279, 2002.

[30] H. Xu and B. Li, "Seen as stable marriages," in *Proc. IEEE INFOCOM*, 2011.

[31] ——, "Egalitarian stable matching for VM migration in cloud computing," in *Proc. IEEE INFOCOM Workshop on Cloud Computing*, 2011.

[32] B. Sotomayor, R. Montero, I. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Comput.*, vol. 13, no. 5, pp. 14–22, September 2009.

[33] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *Proc. IEEE CNSM*, 2010.

[34] A. Kalbasi, D. Krishnamurthy, J. Rolia, and M. Richter, "MODE: Mix driven on-line resource demand estimation," in *Proc. IEEE CNSM*, 2011.

[35] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. ACM SoCC*, 2011.

Hong Xu received his B.Engr. degree from the Department of Information Engineering, The Chinese University of Hong Kong, in 2007 and the M.A.Sc. degree from the Department of Electrical and Computer Engineering, University of Toronto. He is now a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Toronto. His research interests include cloud computing, network economics, and wireless networking. His current focus is on resource management in cloud computing using economics and optimization theory. He is a student member of IEEE.

Baochun Li received the B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000.

Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Nortel Networks Junior Chair in Network Architecture and Services from October 2003 to June 2005, and the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks.

Dr. Li was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a member of ACM and a senior member of IEEE.